Kevin Knorr

Auswahl, Konzeption und Implementierung eines geeigneten Verfahrens zur Pfadberechnung und Kollisionsvermeidung

eingereicht als

BACHELORARBEIT

an der

HOCHSCHULE MITTWEIDA

UNIVERSITY OF APPLIED SCIENCES

Fachbereich Informatik

Mittweida, 16. September 2011

Hochschulbetreuer: Prof. Dr.-Ing. M. Geißler
Firmenbetreuer: Dipl.-Inf. T. Jäckel

Bibliographische Beschreibung:

Knorr, Kevin:
Auswahl, Konzeption und Implementierung eines geeigneten Verfahrens zur Pfadberechnung und Kollisionsvermeidung. - 2011. - 60 S. Mittweida, Hochschule Mittweida, Fachbereich Informatik, Bachelorarbeit, 2011

Referat:

Das Ziel der Arbeit besteht in der Auswahl, Konzeption und Implementierung eines geeigneten Verfahrens zur Pfadberechnung und Kollisionsvermeidung in einem 3D-Simulations-System zur Planung und Visualisierung menschlicher Arbeit. Dabei werden verschiedene Datenstrukturen und Algorithmen anhand ihrer Laufzeit und ihres Speicherbedarfs untersucht. Im Anschluss werden die für geeignet befundenen Datenstrukturen und Algorithmen für die Implementierung eines Systems zur Pfadfindung verwendet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

AABB Axis Aligned Bondunding Box

BS Bounding Sphere

BSP Binary Space Partitioning

BV Bounding Volume

EAWS European Assembly Worksheet

ema Editor menschliche Arbeit

kD k-dimensional

MTM Methods-Time Measurement

OBB Oriented Bounding Box

PLM Product Lifecycle Management

SAT Separating Axis Theorem

1 Einleitung

Die vorliegende Arbeit beschäftigt sich mit der Aufgabe, einen optimalen Weg in einer Umgebung mittels Pfadfindung zu ermitteln.

1.1 Motivation

In der heutigen Zeit werden die Rechnersysteme immer schneller und leistungsfähiger. Besonders die 3D Grafik entwickelt sich immer rasanter. Computer werden in allen Bereichen der Produktentwicklung eingesetzt. Daher ist es umso verwunderlicher, dass zahlreiche Planer immer noch manuell arbeiten, indem sie Arbeitsabläufe in Tabellenform planen. Dazu wird allerdings ein hohes Maß an Vorstellungskraft vom Planer vorausgesetzt, da er keine Möglichkeit hat, seine Planung visuell zu validieren. Dieser Umstand bringt ein hohes Fehlerrisiko mit sich, welches mit einer Sichtprüfung leicht zu beheben ist. Es existiert bereits Software zum Visualisieren menschlicher Tätigkeit. Bei dieser müssen jedoch die Bewegungen Pose für Pose erstellt werden. Dieses Verfahren ist allerdings Zeit aufwendig und dadurch teuer.

Die Chemnitzer Firma imk automotive GmbH, deren Hauptkompetenz in der Fahrzeug- und Fertigungsprozessentwicklung liegt, hat die Vison, die Digitale Fabrik zu vervollständigen. Das heißt, es soll möglich sein, ein virtuelles Abbild eines Produktes – zum Beispiel eines Fahrzeugs – in einer Simulation zu fertigen, bevor die reale Umsetzung beginnt. Den Schwerpunkt setzt die imk automotive GmbH dabei auf die Simulation menschlicher Tätigkeit.
Die daraus entstandene Software heißt „Editor menschliche Arbeit“ (ema). Sie ist eine modular aufgebaute Software, die sich dadurch an verschiedene Product Lifecycle Management (PLM) Systeme ankoppeln kann. Derzeit wird „DELMIA“ – eine Software, welche die Planung, Visualisierung, Simulation und Absicherung in der Produktion ermöglicht – als PLM System unterstützt.

Abbildung 1: ema Logo

Der ema ist ein Werkzeug, welches durch Algorithmen gesteuert eine Simulation menschlicher Arbeit erzeugt und diese mit verschiedenen Planungsmethoden[1] verknüpft. Ziel ist es, mit möglichst wenigen Eingaben eine korrekte Simulation des Prozesses zu erstellen. Diese Simulationen können dann zur Analyse des Arbeitsplatzes verwendet werden – unabhängig davon, ob dieser Arbeitsplatz bereits real existiert oder sich noch in der Planung befindet. Ebenfalls ist es möglich, die Simulationen für Schulungen und Präsentationen zu verwenden.
Allerdings hängt der Erfolg des ema von der Glaubwürdigkeit der Ergebnisse und der Einfachheit der Bedienung ab. Zur Glaubwürdigkeit gehört, dass die Bewegungen, die der virtuelle Werker ausführt, biomechanisch korrekt sind und den Bewegungen eines realen Werkers ähneln. Außerdem muss sich die Software auch an physikalische Gesetze halten. So darf der Werker nicht durch die Luft fliegen oder durch Objekte laufen. Zur Einfachheit der Bedienung gehört, dass so wenige Benutzereingaben wie möglich verlangt und stattdessen so viele Daten wie möglich automatisch generiert werden – wie z.B., dass der virtuelle Werker automatisch einen Weg von seiner aktuellen Position zu seiner Arbeitsposition findet.
Mit dem ersteren Problem, der biomechanisch korrekten Bewegungsgenerierung, beschäftigt sich das Projekt „eMAN“. In der vorliegenden Arbeit wird das Augenmerk speziell auf die Problematik gelegt, Pfade aufzufinden, ohne sich durch Objekte zu bewegen.

Abbildung 2: eMAN Logo

[1] z.B. Methods-Time Measurement (MTM)-Analyse, European Assembly Worksheet (EAWS)-Ergonomiebewertung

1.2 Aufgabenstellung

Der ema befindet sich momentan in der ersten Version. Somit wurden bereits alle Grundfunktionalitäten implementiert. Es existieren Funktionen, welche eine Pfadfindung realisieren. Diese Funktionen basieren auf einem regelmäßigen Gitter als Datenstruktur und dem A*-Algorithmus[2] für die Pfadfindung. Sie werden für die Berechnung von Laufwegen und Bahnen für Körperteile verwendet.
Die Generierung der Pfade ist aus Zeitgründen bei der bisherigen Entwicklung recht einfach gehalten und nicht optimal implementiert. Für die Weiterentwicklung ist es allerdings notwendig, die Generierung der Pfade hinsichtlich Speicherbedarf und Rechenzeit zu optimieren. Weiterhin ist es mit dem aktuellen Konzept nicht möglich, Kollisionen mit beweglichen Objekten zu verhindern.

Die Aufgabe des Autors ist es, ein Konzept zu entwickeln, welches Datenstrukturen und Algorithmen für eine effiziente Pfadsuche beschreibt. Dazu sind verschiedene Datenstrukturen und Algorithmen zu suchen und zu testen. Dabei werden die Datenstrukturen hinsichtlich ihres Speicherplatzverbrauchs untersucht, während die Algorithmen bezüglich ihrer Laufzeit analysiert werden. Außerdem werden Möglichkeiten zum Betrachten dynamischer Objekte getestet.
Am Ende der Arbeit ist das Konzept prototypisch umzusetzen und auf Praxistauglichkeit zu evaluieren.

1.3 Gliederung der Arbeit

Im Kapitel 2 wird grundlegendes Wissen vermittelt, das mit der Problematik der Pfadfindung und der Graphentheorie zusammenhängt. Hier werden Begriffe wie z.B. Digitale Fabrik, Raumunterteilung, Kollisionserkennung und Hüllkörper näher beleuchtet. Im 3. Kapitel „Lösungskonzepte“ werden die wichtigsten Datenstrukturen und Algorithmen, die zur Implementierung des Prototyps in Frage kommen, betrachtet und einander gegenüber gestellt. Anschließend folgt in Kapitel 4 die Implementierung dieser Datenstrukturen und Algorithmen. An dieser Stelle wird anhand von verschiedenen Diagrammtypen ein Überblick über das Testsystem gegeben sowie mithilfe von Quellcodeausschnitten die Funktionsweise erläutert. Eine Untersuchung der implementierten Datenstrukturen und Algorithmen hinsichtlich ihrer Rechengeschwindigkeit und ihres Speicherbedarfs folgt im 5. Kapitel. Anhand der Ergebnisse der Untersuchung werden im Kapitel 6 die Implementierungsdetails des Prototyps beschrieben.

[2] sprich: A-Stern-Algorithmus

2 Grundlagen

In diesem Kapitel werden wichtige Grundlagen gelegt, die zum Verständnis der Arbeit benötigt werden, sowie zusätzliche Hintergrundinformationen gegeben.

2.1 Digitale Fabrik

Die Digitale Fabrik ist eine Sammlung von Modellen und Methoden zur ganzheitlichen Planung, Realisierung, Steuerung und Verbesserung aller Prozesse der Produktion.
Sie entstand aus den Bemühungen der Produktionstechnik, die Computerintegration zu verbessern. Das Ziel der Digitalen Fabrik ist es, den gesamten Entstehungsprozess eines Produktes virtuell abzubilden – von der Architektur der Werkhalle über die Arbeitsplatzgestaltung bis zur Arbeitsablaufplanung.
Mit Hilfe der Digitalen Fabrik können die meisten Probleme, welche bei der Produktion auftreten könnten, im Vorfeld untersucht, erkannt und behoben werden.
Weiterführende Informationen gibt die VDI-Richtlinie [VR08].

Abbildung 3: Digitale Werkhalle mit Menschmodellen

2.2 Editor menschliche Arbeit (ema)

Der ema, welcher von der Firma imk automotive GmbH in Chemnitz entwickelt wird, ist eine Software zur Ergänzung der Digitalen Fabrik. Die Grundlage der Software entstand aus der Diplomarbeit von Sebastian Bauer (vgl. [Bau06]). Der ema enthält ein 3D Menschmodell zur Planung, Simulation, Gestaltung und ergonomischen Analyse menschlicher Arbeit. Das Ziel der Software ist es, dem 3D Menschmodell erweiterte Fähigkeiten zu übertragen, die einem realen Werker möglichst ähnlich sind.
Zwar exsistieren bereits verschiedene Menschmodelle, diese enthalten aber keine eigene Intelligenz. Für jede Pose ist jedes einzelne Gelenk richtig auszurichten. Da für eine Animation sehr viele Posen benötigt werden, ist es zwar möglich, mit den schon vorhandenen Menschmodellen Simulationen zu erstellen, aber diese Möglichkeit ist sehr zeitaufwendig und damit teuer.

Ziel des ema ist es, mit so wenigen Benutzereingaben wie möglich eine Simulation zu erstellen. Der Benutzer soll keine Einzelbewegungen mehr modellieren müssen. Er kann mit Hilfe von Verrichtungen[3] Prozesse modelieren und gestalten. Diese Verrichtungen sind einzelne Arbeitsschritte, wie z.B. „Gehe zum Ziel“ oder „Aufnehmen und Ablegen“. Durch Angabe von Parametern zu den einzelnen Verrichtungen können diese nach Benutzerwünschen angepasst werden. Aus diesen parametrisierten Verrichtungen erzeugt der im ema enthaltene Bewegungsgenerator[4] die für die Simulation benötigten Posen.
Mit diesen Bewegungsdaten wird zunächst intern eine Sollzeit-Analyse nach Methods-Time Measurement (MTM) und eine Ergonomiebewertung nach European Assembly Worksheet (EAWS) durchgeführt.
Die Sollzeit-Analyse berrechnet die notwendige Zeit, die ein Werker für die Durchführung der Arbeit benötigt und stützt sich dabei auf die Methoden der MTM.
Die Ergonomiebewertung stuft die Qualität des Arbeitsplatzes und die Belastung des Werkers ein. Dazu wird auf das Punktesystem des EAWS zurückgegriffen.
Diese zwei Analysen werden nach der Berechnung direkt im ema angezeigt. Zudem kann danach die Simulation des Prozesses gestartet werden, um eine Sichtprüfung durchzuführen.

[3] aus einer Verrichtungsbibliothek
[4] ebenfalls schon implementiert, aber im „eMAN“-Projekt weiterentwickelt

2.3 Hüllkörper (bounding volumes)

Hüllkörper oder auch *bounding volumes (BV)* sind einfache geometrische Körper, welche ein komplexes geometrisches Objekt umschließen. Sie werden eingesetzt um verschiedene Algorithmen – wie zum Beispiel Kollisionserkennung oder *Raytracing* – zu beschleunigen, da sich einfache geometrische Körper leichter testen lassen. In dem Vortrag von Mario Amrehn werden weiterführende Informationen vermittelt (vgl. [Amr09]).

Gängige Hüllkörper sind:

- Hüllkreis bzw. -kugel

Abbildung 4: Das Framo Logo innerhalb eines Hüllkreises

Eine Hüllkugel oder auch *bounding sphere (BS)* ist ein spezieller Hüllkörper in der Form einer Kugel im dreidimensionalen Raum bzw. eines Kreises im zweidimensionalen Raum. Ihr Mittelpunkt sollte möglichst mit dem Mittelpunkt des Objekts übereinstimmen, das die Hüllkugel umschließt. Sie wird durch Mittelpunkt und Radius eindeutig bestimmt.

- Achsenausgerichtetes Hüllrechteck bzw. -quader

Abbildung 5: Das Framo Logo innerhalb eines achsenausgerichteten Hüllrechtecks

Ein achsenausgerichteter Hüllquader oder auch *Axis Aligned Bondunding Box (AABB)* ist eine spezielle BV, die die Form eines Quaders (3D) bzw. eines Rechtecks (2D) aufweist, bei denen die Kanten parallel zu den Koordinatenachsen verlaufen. Sie kann eindeutig durch die Minimal- und Maximalwerte auf den Achsen bestimmt werden.

- Ausgerichtetes Hüllrechteck bzw. -quader

Abbildung 6: Das Framo Logo innerhalb eines ausgerichteten Hüllrechtecks

Ein ausgerichteter Hüllquader oder auch *Oriented Bounding Box (OBB)* ist dadurch gekennzeichnet, dass sie analog zur AABB die Form eines Quaders bzw. Rechtecks aufweist. Allerdings verlaufen die Kanten bei einer OBB nicht parallel zu den Achsen des Koordinatensystems, stattdessen ist sie mit dem Objekt gedreht. Der ausgerichtete Hüllquader wird eindeutig durch ihre Eckpunkte bestimmt.

2.4 Kollisionserkennung (collision detection)

Mittels Kollisionserkennung oder auch *collision detection* ist es möglich, einander berührende oder überlappende Objekte im Raum zu erkennen. Sie findet Verwendung in Simulationen und Computerspielen. Die Autoren Jens Schedel, Christoph Forman und Philipp Baumgärtel beschäftigen sich mit dieser Thematik in ihrem Vortrag [SFB05].
Kollisionserkennung wird meist in zwei Phasen implementiert. Einmal eine grobe, aber dafür schnelle Phase, um schon einige Fälle auszuschließen. Danach kann – wenn benötigt – in einer zweiten Phase eine genauere Untersuchung stattfinden, welche dementsprechend mehr Rechenzeit benötigt.
In der ersten Phase wird meist ein Hüllkreis bzw. eine Hüllkugel um die zu betrachtenden Objekte gelegt. Diese können ohne größeren Rechenaufwand durch Vergleichen des Abstandes der Mittelpunkte und der Radien getestet werden. Falls diese Überprüfung keine Kollision ermittelt, ist das Auftreten einer Kollision ausgeschlossen und die Untersuchung braucht nicht fortgesetzt zu werden. Andernfalls ist eine nähere Betrachtung in einer zweiten Phase notwendig. Ein oft verwendeter Algorithmus für genaue Berechnungen ist das Separating Axis Theorem (SAT). Dieses geht davon aus, dass für den Fall, dass eine separierende Achse existiert, keine Kollision stattfindet. Eine separierende Achse ist eine Gerade, welche die Objekte voneinander trennt.

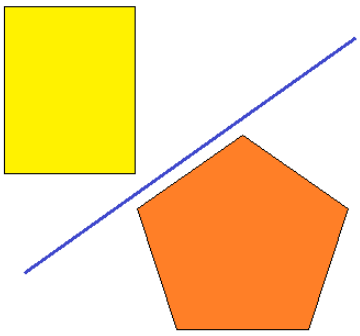

Abbildung 7: Darstellung einer separierenden Achse (blau) zwischen zwei Objekten (gelb und orange)

2.5 Szenegraph

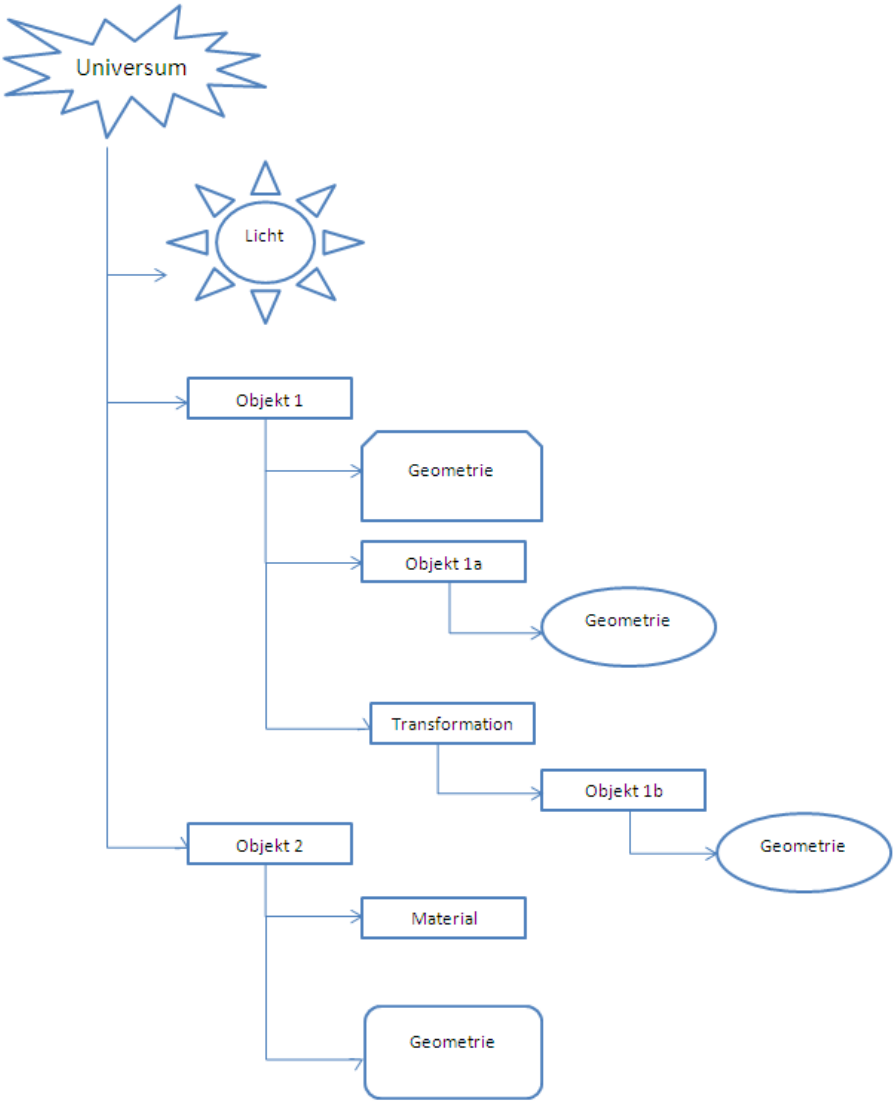

Abbildung 8: ein einfacher Beispielszenengraph

Der Szenengraph ist eine objektorientierte Datenstruktur, die aus Knoten und Kanten besteht. Er wurde für computergrafische Anwendungen entwickelt und enthält alle Informationen zu einer Szene, wie z.B. die räumliche Anordnung aller Objekte. Ein Vortrag von Jens-Fabian Goetzmann beschäftigt sich mit diesem Thema (vgl. [Goe06]).
Aus graphentheoretischer Sicht kann ein Szenengraph als zusammenhängender, einseitig gerichteter, azyklischer Graph und aus informationstechnicher Sicht

als Baum betrachtet werden. Dabei enthält die Wurzel die gesamte Szene – das Universum. Alle Kindelemente[5] können je nach Verwendungsart verschiedene Objekte – wie zum Beispiel Geometrie, Material, Licht, Viewpoints – enthalten. Somit ist ein Szenengraph auch eine hierarchiche Ordnung der Objekte.
Jeder Knoten enthält zusätzlich eine Transformationsmatrix[6], welche die Position relativ[7] zum übergeordneten Knoten angibt. Eine Umwandlung in absolute Koordinaten[8] ist durch die Addition der Transformationmatritzen auf dem Weg zur Wurzel des Szenengraphs möglich.
Ein Szenengraph wird oft für die Optimierung von Kollisionsabfragen oder zum effizienten Rendern verwendet. Außerdem erleichtert er das Verändern der Szene, da eine Änderung der Position eines Knotens auch die Positionen der Kindknoten verändert. So können zum Beispiel Objekte, die semantisch zusammengehören unter einem Knoten zusammengefasst und dann miteinander bewegt werden.

2.6 Raumunterteilung (space partioning)

Um die oben genannten Algorithmen umzusetzen, muss von der Umgebung zunächst eine logische Repräsentation erzeugt werden. Dies geschieht mithilfe der Raumunterteilung oder auch *space partitioning*, wie in [Ale07] von Marc Alexa beschrieben. Dabei wird der Raum zunächst in Teilbereiche zerlegt, die sich nicht überlappen dürfen. Dadurch kann jeder Punkt im Raum eindeutig einem Teilbereich zugeordnet werden. Jeder dieser Teilbereiche hat einen Status: Er kann z.B. begehbar oder blockiert sein. Damit die logische Repräsentation von den Pfadfindungsalgorithmen verwendet werden kann, wird sie in einen kantengewichteten Graphen überführt. Die begehbaren Teilbereiche werden in Knoten transformiert und durch Kanten miteinander verbunden, falls ein Weg zwischen den Teilbereichen existiert.

Gängige Systeme zur Raumunterteilung sind:

- uniform Grids[9]
- Quad-Tree(2D) / Oct-Tree(3D)
- k-dimensionaler Tree (kD-Tree)
- Binary Space Partitioning-Tree (BSP-Tree)

[5] sowohl Blätter als auch innere Knoten
[6] enthält Translation, Rotation und Skalierung
[7] Objektkoordinaten
[8] Weltkoordinaten
[9] gleichmäßiges Raster

2.7 Pfadfindung

Die Pfadfindung beschäftigt sich mit der Aufgabe, einen optimalen Weg zwischen zwei Punkten in einer bekannten Umgebung zu ermitteln. Dieser Schwerpunkt wird von Daniel Kastenholz in seinem Vortrag über *Pathfinding* aufgegriffen (vgl. [Kas06]).

Um diese Problematik zu lösen, greift sie auf die Möglichkeiten der Graphentheorie zurück. Dabei wird die Umgebung auf einen Graphen projiziert. Die Knoten in einem Graphen stellen dabei erreichbare Punkte in der Umgebung dar. Die Kanten zwischen den Knoten des Graphen repräsentieren die einzelnen Wege zwischen den entsprechenden Punkten.

Als Gewichtung der Kanten können verschiedene Qualitätsmerkmale der Wege verwendet werden, wie z.B. die Länge des Weges, die benötigte Zeit oder den Aufwand. Die Summe der Gewichtungen der auf einem Weg liegenden Kanten wird als Wegekosten bezeichnet. Der optimale Weg ist der Weg zwischen zwei Punkten, der die geringsten Wegekosten besitzt. Um diesen Weg zu bestimmen, gibt es verschiedene Möglichkeiten. Diese können durch Algorithmen wie Breitensuche, Tiefensuche, Dijkstra- oder A*-Algorithmus implementiert werden. Mithilfe dieser Algorithmen ist es möglich, den optimalen Weg zwischen zwei Punkten zu bestimmen.

3 Lösungskonzepte

Der kürzeste Weg zwischen zwei Knoten kann mit Hilfe von Pfadfindungsalgorithmen ermittelt werden. Um einen solchen Pfadfindungsalgorithmus zu implementieren, ist ein Wegeplan erforderlich. Durch sinnvolle Raumunterteilung kann ein solcher Wegeplan erzeugt werden. Dabei ist jedoch eine Kollisionserkennung notwendig, um festzustellen, ob ein Teilbereich begehbar ist oder ob sich ein Kollisionsobjekt darin befindet. Damit es möglich ist, Laufzeit und Speicherbedarf zu minimieren, wird eine gute Kombination aus Datenstruktur und Algorithmus benötigt. Um eine derartige Kombination zu ermitteln, werden die Datenstrukturen und Algorithmen zunächst im zweidimensionalen Raum in einer dafür entwickelten Umgebung getestet. Anschließend wird die optimale Kombination im dreidimensionalen Raum für den ema implementiert.
Dabei werden zunächst folgende Datenstrukturen testweise implementiert und evaluiert:

- Raster
- QuadTree

Folgende Algorithmen werden zur Implementierung verwendet:

- Algorithmus von Dijkstra
- A*-Algorithmus

Auf diese Datenstrukturen und Algorithmen wird im Folgenden genauer eingegangen.

3.1 Datenstrukturen

Für die Pfadfindung werden „Wegpunkt- Graphen“ benötigt. Da jedoch die Umgebungsdaten erst zur Laufzeit vorliegen, ergibt sich das Problem, dass keine Vorberechnung möglich ist. Deshalb ist es notwendig, den Raum und seine Objekte zu analysieren, um einen „Wegpunkt- Graphen“ zu berechnen. Um dies zu ermöglichen, wird der Raum in sich nicht überlagernde Teilbereiche unterteilt und in Datenstrukturen gespeichert. Dies gewährleistet die eindeutige Zuordnung eines Punktes im Raum zu einem der Teilbereiche. Die Zentren der Teilbereiche ohne Kollisionsobjekte stellen die Knoten des „Wegpunkt- Graphs“ dar, während die Nachbarschaftsbeziehungen seinen Kanten entsprechen. Die Kantengewichte berechnen sich aus der Entfernung der Knoten des Graphs, also der Zentren der Teilbereiche.

3.1.1 Gleichförmiges Raster (Uniform Grid)

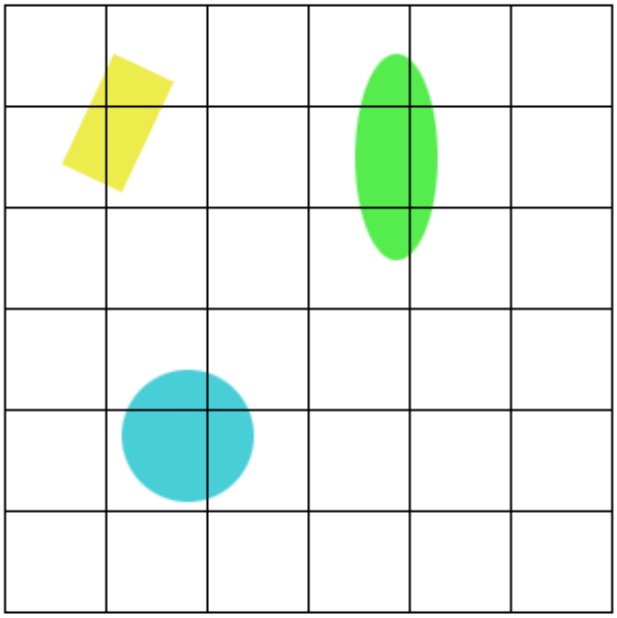

Abbildung 9: Beispiel für ein Raster mit Objekten

Ein gleichförmiges Raster ist eine nicht hierarchische Datenstruktur, die aus gleich großen, zumeist quadratischen Zellen besteht. Dabei wird jedes Objekt mindestens einer Zelle zugeordnet. Die Vorteile dieser Datenstruktur liegen in der einfachen Implementierung und der Möglichkeit eines schnellen Updates. Dazu kommt, dass das Raster effizient und einfach traversierbar und raumzentrisch ist. Das Raster kann sich jedoch nicht der Geometrie anpassen und ist sehr speicherlastig. Da mehrere Zellen dasselbe Objekt enthalten können, entsteht zudem Objektredundanz.

3.1.2 2^n Trees (QuadTree(2D) / OctTree(3D))

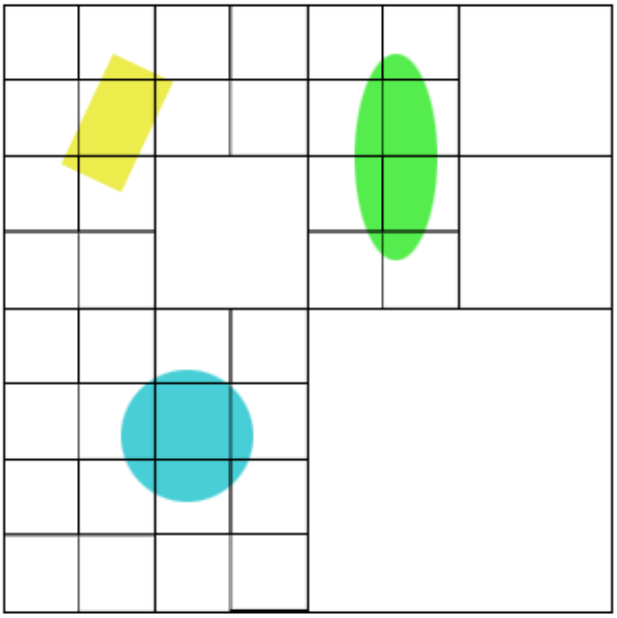

Abbildung 10: Beispiel für ein QuadTree mit Objekten

Diese ebenfalls raumzentrischen Datenstrukturen sind baumartig aufgebaut. Im Gegensatz zum Raster sind sie hierarchisch. Sie zerlegen den Raum rekursiv in 2^n gleich große Zellen, wobei n die Anzahl der Dimensionen. 2^n Trees haben ebenso wie Raster den Nachteil, dass Objektredundanz auftreten kann. Verglichen mit dem Raster haben sie jedoch weniger Speicherbedarf und erlauben eine bessere Anpassung an die Geometrie. Das Traversieren durch den Baum und die Suche von benachbarten Zellen gestaltet sich jedoch komplizierter.

3.2 Algorithmen

Mithilfe der folgenden Algorithmen ist es möglich, den günstigsten Weg zwischen zwei Knoten in einem kantengewichteten Graphen zu finden.

3.2.1 Algorithmus von Dijkstra

Dieser Greedy Algorithmus wurde nach seinem Erfinder Edsger W. Dijkstra benannt. Der Algorithmus arbeitet in einem positiv-kantengewichteten Graphen und wird verwendet, um die kürzesten Pfade bei einem vorgegebenen Startknoten zu ermitteln. Der Wert eines Knotens gibt dabei die Länge des kürzesten Weges vom Start- bis zu dem aktuellen Knoten an. So wird genau die Kante verfolgt, die den kürzesten Weg vom Startknoten verspricht. In einer offenen Liste werden alle noch zu besuchenden Knoten gespeichert. Grob betrachtet läuft der Algorithmus in zwei Schritten ab: Zuerst werden die Werte aller Knoten mit unendlich sowie die Vorgänger aller Knoten mit null initialisiert. Anschließend wird, solange die offene Liste nicht leer ist, zunächst der Knoten mit dem kleinsten Wert aus der offenen Liste ausgewählt sowie alle nicht besuchten Nachbarn dieses Knotens ermittelt. Dabei werden die Werte wie folgt aktualisiert: Ist der neue Wert kleiner als der alte, wird der Nachbarknoten auf diesen Wert gesetzt und der aktuelle Knoten als Vorgänger gespeichert. Der kürzeste Weg wird ermittelt, indem der Weg vom Zielknoten über die jeweiligen Vorgänger bis zum Startknoten verfolgt wird.

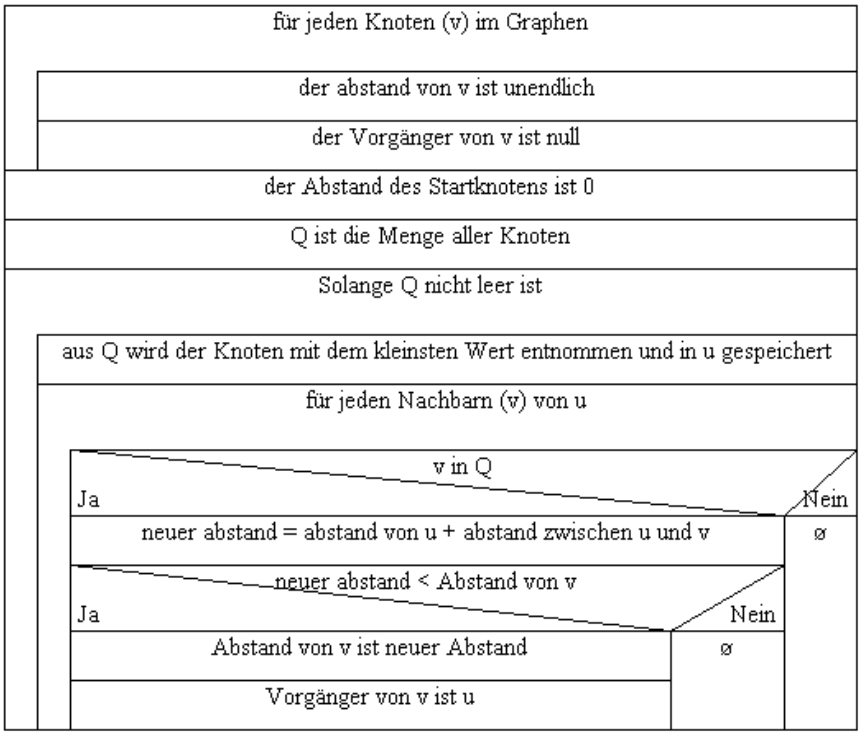

Abbildung 11: Der Algorithmus von Dijkstra als Nassi-Shneiderman-Diagramm

3.2.2 A*-Algorithmus

Der A*-Algorithmus, ein informierter Suchalgorithmus, berechnet einen der kürzesten Pfade zwischen zwei Knoten in einem positiv-kantengewichteten Graphen. Er stellt die Erweiterung des Dijkstra-Algorithmus dar, da er eine Schätzfunktion (Heuristik) implementiert. Unter einer Heuristik wird eine Funktion verstanden, welche mit geringem Aufwand eine annähernd genaue Lösung erzeugt. Im Fall der Pfadfindungsalgorithmen kann die Länge des direkten Pfades zwischen zwei Punkten als einfache Heuristik verwendet werden. Damit besteht die Möglichkeit, im Graphen zielgerichtet zu suchen, wodurch die Laufzeit verkürzt wird. Der A*-Algorithmus garantiert, dass immer die optimale Lösung gefunden wird, falls eine existiert.
Im Gegensatz zum Dijkstra-Algorithmus wird in den Knoten zusätzlich zum zurückgelegten Weg ebenso die Heuristik gespeichert. Damit wird immer der Weg verfolgt, der die kürzeste Entfernung vom Start- bis zum Zielknoten verspricht.

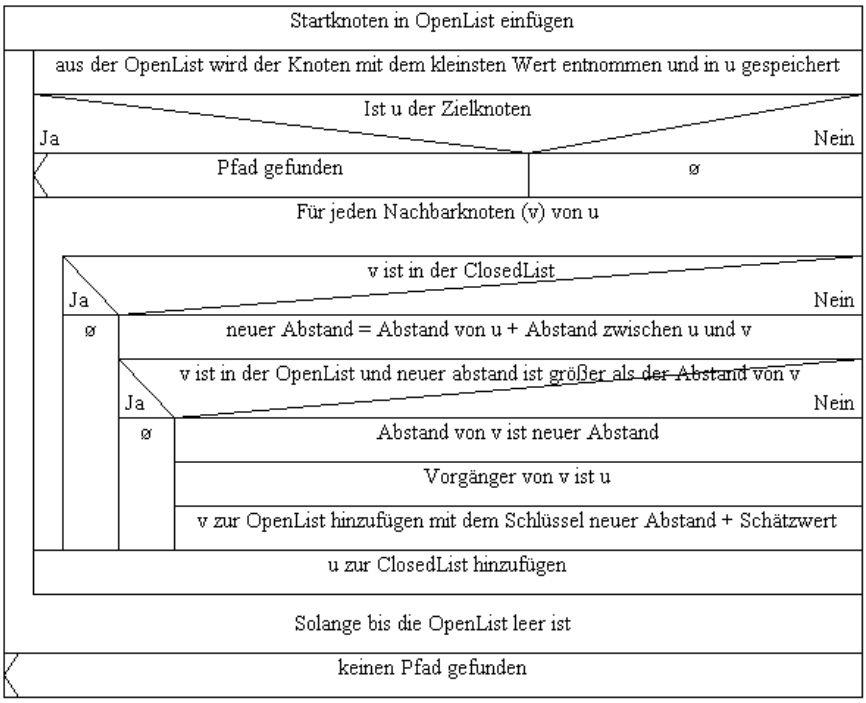

Abbildung 12: Der A*-Algorithmus als Nassi-Shneiderman-Diagramm

4 Implementierung

In diesem Kapitel werden die wichtigsten Implementierungsdetails anhand von Quellcodeausschnitten erläutert. Zusätzlich werden verschiedene Diagramme eingesetzt, um den Aufbau und den Ablauf der Software zu demonstrieren.

Als Programmiersprache zur Implementierung wird C# eingesetzt, da die Software ema ebenfalls in dieser Sprache entwickelt wurde und daher der Prototyp als Klassenbibliothek eingebunden werden kann.

4.1 Komponentendiagramm

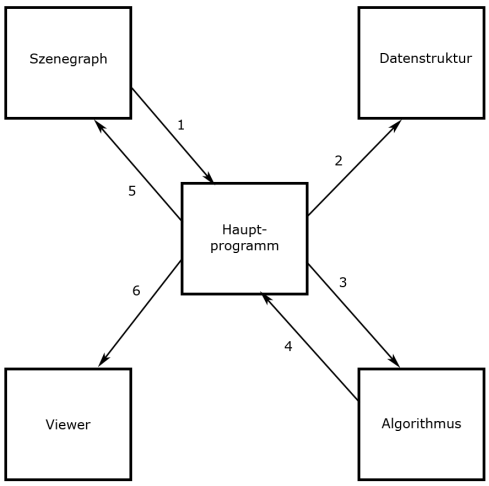

Abbildung 13: Komponentendiagramm

Abbildung 13 zeigt den groben Aufbau des Systems. Die Pfeile haben dabei folgende Bedeutung:

1 – Entnimmt Bodenobjekte und Kollisionsobjekte

2 – Initialisiert und übergibt Boden- und Kollisionsobjekte

3 – Initialisiert und übergibt Datenstruktur

4 – Liefert Pfad

5 – Übergibt Datenstruktur und Pfad

6 – Initialisiert und übergibt Szenegraphen

4.2 Klassendiagramme

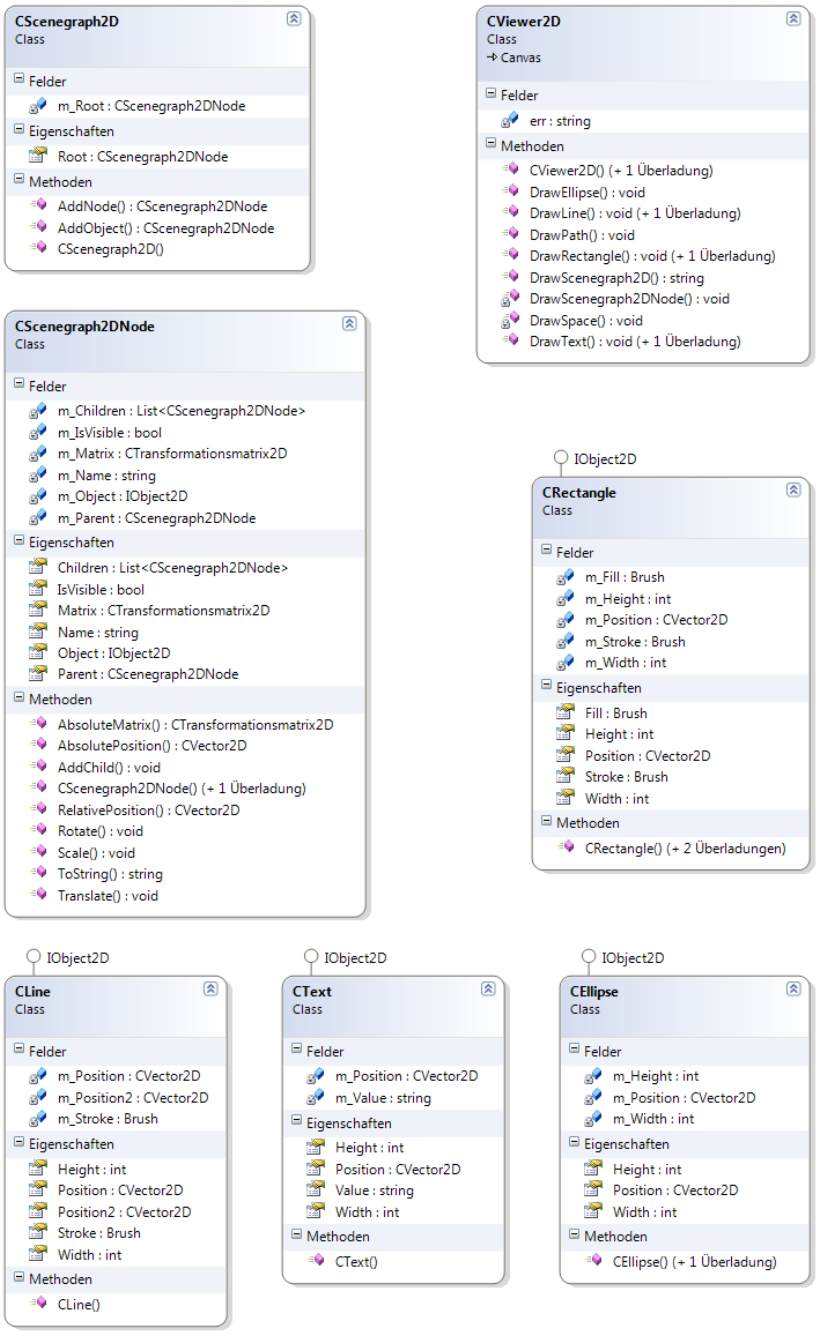

Abbildung 14: Klassendiagramm der Klassen, die für den Viewer benötigt werden.

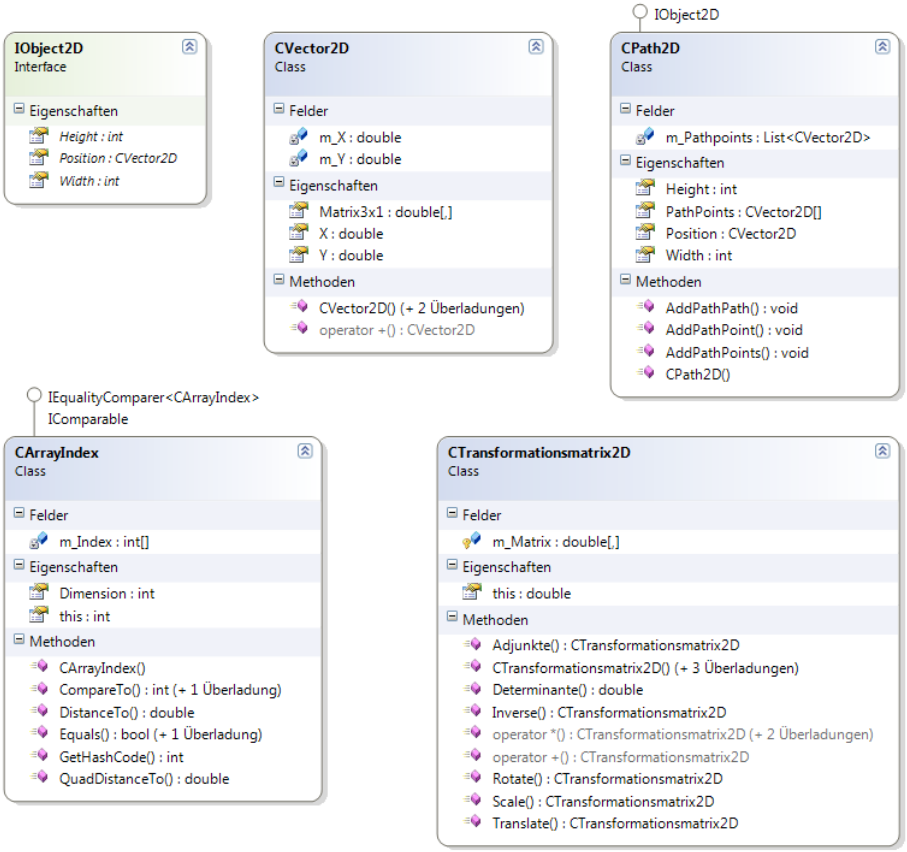

Abbildung 15: Klassendiagramm der Hilfsklassen.

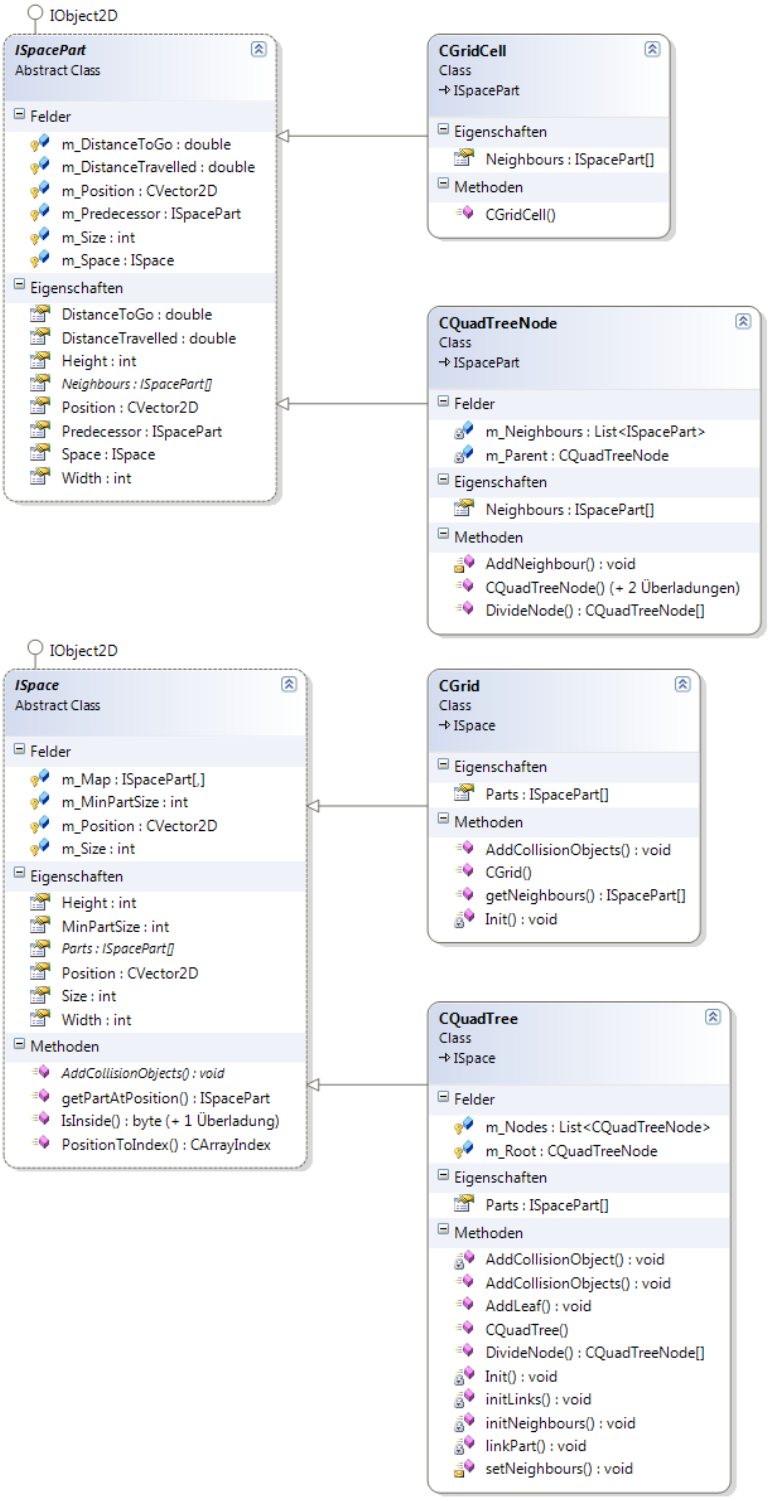

Abbildung 16: Klassendiagramm der Klassen, die zu den Datenstrukturen gehören.

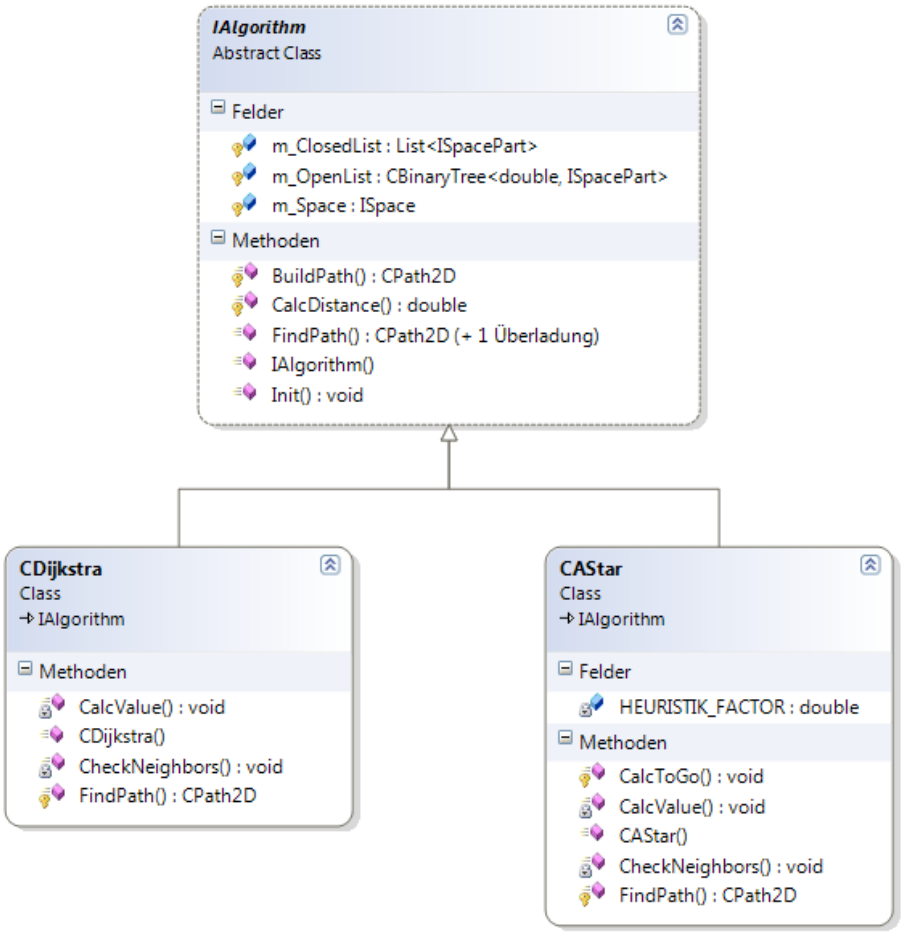

Abbildung 17: Klassendiagramm der Klassen, die zu den Algorithmen gehören.

4.3 Implementierung der Datenstrukturen

Jede Datenstruktur besteht aus zwei Teilen: der eigentlichen Datenstruktur sowie den einzelnen Teilbereichen. Alle Datenstrukturklassen sind von einer abstrakten Klasse abgeleitet.
Diese ist wie folgt aufgebaut:

```
1  public abstract class ISpace : IObject2D
2  {
3      protected CVector2D m_Position;
4
5      public CVector2D Position
6      {
7          get { return m_Position; }
8          set { m_Position = value; }
9      }
10
11     protected int m_Size;
12
13     public int Size
14     {
15         get { return m_Size; }
16         set { m_Size = value; }
17     }
18
19     public int Width
20     {
21         get { return m_Size; }
22         set { m_Size = value; }
23     }
24
25     public int Height
26     {
27         get { return m_Size; }
28         set { m_Size = value; }
29     }
30
31     protected int m_MinPartSize;
32
33     public int MinPartSize
```

```
34      {
35          get { return m_MinPartSize; }
36      }
37
38      protected ISpacePart[,] m_Map;
39
40      public abstract ISpacePart[] Parts { get; }
41
42      public ISpacePart getPartAtPosition(CVector2D
            iPosition)
43      {
44          CArrayIndex ai = PositionToIndex(iPosition);
45          return m_Map[ai[0], ai[1]];
46      }
47
48      public CArrayIndex PositionToIndex(CVector2D
            iPosition)
49      {
50          int i, j;
51          i = (int)Math.Abs(iPosition.X - (m_Position.X
                - (double)m_Size / 2.0)) / m_MinPartSize;
52          j = (int)Math.Abs(iPosition.Y - (m_Position.Y
                - (double)m_Size / 2.0)) / m_MinPartSize;
53          return new CArrayIndex(i, j);
54      }
55
56      public abstract void AddCollisionObjects(IObject2D
            [] iCollisionObjects);
57
58  }
```

Listing 1: Abstrakte Klasse `ISpace`

Die Klasse implementiert das Interface `IObject2D` und dessen Member für Position und Größe der repräsentierten Fläche. Sie enthält Hilfsmethoden, die für die Erstellung der Datenstruktur notwendig sind und deklariert Methoden, die alle Datenstrukturen implementieren müssen.

Die Teilbereiche der Datenstrukturen leiten sich von der Klasse `ISpacePart` ab.

```
1  public abstract class ISpacePart : IObject2D
2  {
3      protected ISpace m_Space;
4
5      public ISpace Space
6      {
7          get { return m_Space; }
8      }
9      protected CVector2D m_Position;
10
11     public CVector2D Position
12     {
13         get { return m_Position; }
14         set { m_Position = value; }
15     }
16     protected int m_Size;
17
18     public int Width
19     {
20         get { return m_Size; }
21         set { m_Size = value; }
22     }
23
24     public int Height
25     {
26         get { return m_Size; }
27         set { m_Size = value; }
28     }
29     protected double m_DistanceTravelled;
30
31     public double DistanceTravelled
32     {
33         get { return m_DistanceTravelled; }
34         set { m_DistanceTravelled = value; }
35     }
36     protected double m_DistanceToGo;
37
```

```
38      public double DistanceToGo
39      {
40          get { return m_DistanceToGo; }
41          set { m_DistanceToGo = value; }
42      }
43      protected ISpacePart m_Predecessor;
44
45      public ISpacePart Predecessor
46      {
47          get { return m_Predecessor; }
48          set { m_Predecessor = value; }
49      }
50
51      public abstract ISpacePart[] Neighbours { get; }
52  }
```

Listing 2: Abstrakte Klasse `ISpacePart`

Auch diese abstrakte Klasse implementiert das Interface `IObject2D`. Weiterhin deklariert sie Member, welche von den Pfadfindungsalgorithmen benötigt werden, wie zum Beispiel den zurückgelegten Weg und den Vorgängerknoten.

4.3.1 Raster

Die Klasse `CGrid` repräsentiert die Raumunterteilungsstruktur eines gleichmäßigen Rasters. Dazu erweitert sie die Klasse `ISpace`. Eine einzelne Zelle des Rasters wird durch die Klasse `CGridCell` beschrieben. Diese ist wiederum von `ISpacePart` abgeleitet.

Mithilfe der an den Konstruktor der Klasse übergebenen Bodenobjekten und der minimalen Zellgröße wird die grundlegende Struktur des Rasters initialisiert. Zusätzlich muss für jede Zelle festgelegt werden, ob sie begehbar ist.

```
1  public override void AddCollisionObjects(IObject2D[]
       iCollisionObjects)
2  {
3      foreach (CGridCell cell in m_Map)
4      {
5          foreach (IObject2D obj in iCollisionObjects)
6          {
```

```
7               if (IsInside(obj, cell) > 0)
8               {
9                   cell.DistanceTravelled = -1;
10                  break;
11              }
12          }
13      }
14  }
```

Listing 3: Klasse `CGrid`: `AddCollisionObjects()`- Methode

Dazu werden diejenigen Zellen, die ein Kollisionsobjekt enthalten, als nicht begehbar markiert, indem ihr zurückgelegter Weg auf -1 gesetzt wird, wie im Listing 3 dargestellt. Nachdem das Raster nun initialisiert ist, wird eine Methode benötigt, die benachbarte Zellen ausgibt.

```
1  public ISpacePart[] getNeighbours(CGridCell iCell)
2  {
3      ISpacePart[] erg = new CGridCell[4];
4      CArrayIndex ai = PositionToIndex(iCell.Position);
5      int rc = m_Size / m_MinPartSize;
6      int i, j;
7      i = ai[0];
8      j = ai[1];
9      if (j > 0)  erg[0] = m_Map[i, j - 1];
10     if (i < rc - 1) erg[1] = m_Map[i + 1, j];
11     if (j < rc - 1) erg[2] = m_Map[i, j + 1];
12     if (i > 0)  erg[3] = m_Map[i - 1, j];
13     return erg;
14 }
```

Listing 4: Klasse `CGrid`: `GetNeighbours()`- Methode

Da Referenzen auf die Zellen eineindeutig in einem zweidimensionalen Array gespeichert werden, sind Zellen, die in diesem Array nebeneinander liegen, auch im tatsächlichen Raum benachbart. Wird ein Index des Arrays um 1 erhöht oder verringert, dann wird dadurch eine benachbarte Zelle gefunden.

4.3.2 QuadTree

Die Klasse `CQuadTree` beschreibt die QuadTree-Datenstruktur zur Raumunterteilung. Sie erweitert die abstrakte Klasse `ISpace`. Einzelne Bereiche werden durch die Klasse `CQuadTreeNode` repräsentiert und erweitern deshalb die Klasse `ISpacePart`. Nach der Initialisierung werden die Kollisionsobjekte in die Datenstruktur eingefügt. Dadurch wird der eigentliche Aufbau der Datenstruktur erzeugt.

```
1  public override void AddCollisionObjects(IObject2D[]
       iCollisionObjects)
2  {
3      foreach (IObject2D obj in iCollisionObjects)
4      {
5          CQuadTreeNode[] nl = m_Nodes.ToArray();
6          foreach (CQuadTreeNode node in nl)
7          {
8              AddCollisionObject(obj, node);
9          }
10     }
11     initLinks();
12     initNeighbours();
13 }
14
15 private void AddCollisionObject(IObject2D iObject,
       CQuadTreeNode iNode)
16 {
17
18     if (iNode.DistanceTravelled > 0)
19     {
20         byte ii = IsInside(iObject, iNode);
21         if (ii == 1)
22         {
23             if (iNode.Width <= m_MinPartSize)
24             {
25                 iNode.DistanceTravelled = -1;
26             }
27             else
28             {
29                 CQuadTreeNode[] childs = DivideNode(
```

```
                    iNode);
30                  foreach (CQuadTreeNode node in childs)
31                  {
32                      AddCollisionObject(iObject, node);
33                  }
34              }
35          }
36          else if (ii == 2)
37          {
38              iNode.DistanceTravelled = -1;
39          }
40      }
41  }
```

Listing 5: Klasse `CQuadTree`: `AddCollisionObjects()`- Methode

Um den QuadTree aufzubauen und nicht begehbare Bereiche zu finden, wird für jedes Kollisionsobjekt eine Kollisionserkennung mit jedem Bereich durchgeführt. Falls eine Kollision festgestellt wird, muss der Bereich geteilt werden, wie im Listing 6 gezeigt.

```
1  public CQuadTreeNode[] DivideNode()
2  {
3      CQuadTreeNode[] erg = new CQuadTreeNode[4];
4      int d = (int)m_Size / 4;
5      erg[0] = new CQuadTreeNode(new CVector2D(
           m_Position.X - d, m_Position.Y - d), this);
6      erg[1] = new CQuadTreeNode(new CVector2D(
           m_Position.X + d, m_Position.Y - d), this);
7      erg[2] = new CQuadTreeNode(new CVector2D(
           m_Position.X + d, m_Position.Y + d), this);
8      erg[3] = new CQuadTreeNode(new CVector2D(
           m_Position.X - d, m_Position.Y + d), this);
9      return erg;
10 }
```

Listing 6: Klasse `CQuadTreeNode`: `DivideNode()`- Methode

Die Teilbereiche müssen nach der Unterteilung weiter untersucht werden. Wurde jedoch die minimale Zellgröße erreicht, kann der Bereich nicht weiter unterteilt werden. Er wird als nicht begehbar markiert. Tritt der Fall ein, dass der Bereich

komplett im Kollisionsobjekt liegt, wird er ebenfalls nicht weiter unterteilt, sondern wird als nicht begehbar markiert.

Da das Traversieren und das Finden von Nachbarn in dieser Datenstruktur kompliziert ist, wird im Hintergrund ein zweidimensionales Array gepflegt, welches die Abbildung der Bereiche auf ein gleichmäßiges Raster darstellt.

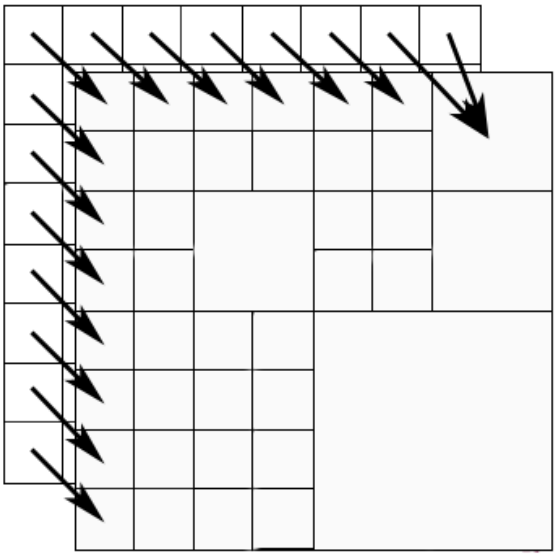

Abbildung 18: Abbildung des QuadTree auf ein Raster

Die Abbildung 18 zeigt, das zweidimensionale Array in Form eines Rasters, welches die Referenzen auf die Teilbereiche des QuadTrees besitzt. Dadurch ist eine einfache Zuordnung von Punkten zu Teilbereichen möglich.

```
1  private void initLinks()
2  {
3      foreach (CQuadTreeNode node in m_Nodes)
4      {
5          linkPart(node);
6      }
7  }
8
9  private void linkPart(CQuadTreeNode iNode)
10 {
```

```
11      CArrayIndex ai = PositionToIndex(new CVector2D(
            iNode.Position.X - (double)iNode.Width / 2.0,
            iNode.Position.Y - (double)iNode.Height / 2.0))
            ;
12      int m = iNode.Width / m_MinPartSize;
13      int n = iNode.Height / m_MinPartSize;
14      for (int i = 0; i < m; i++)
15      {
16          for (int j = 0; j < n; j++)
17          {
18              m_Map[ai[0] + i, ai[1] + j] = iNode;
19          }
20      }
21  }
```

Listing 7: Klasse `CQuadTree`: `LinkPart()`- Methode

Um diese Abbildungstabelle zu erzeugen, wird die Methode `LinkPart()` verwendet. Diese berechnet für jeden Bereich die Stellen des Arrays, die eine Referenz auf den Bereich erhalten.
Diese Abbildungstabelle kann nun zum Auffinden der Nachbarn verwendet werden.

```
1   private void initNeighbours()
2   {
3       foreach (CQuadTreeNode node in m_Nodes)
4       {
5           setNeighbours(node);
6       }
7   }
8
9   internal void setNeighbours(CQuadTreeNode iNode)
10  {
11      int sizeLevel = iNode.Width / m_MinPartSize;
12      CArrayIndex ai = PositionToIndex(new CVector2D(
            iNode.Position.X - (double)iNode.Width / 2.0,
            iNode.Position.Y - (double)iNode.Height / 2.0))
            ;
13      int d = m_Size / m_MinPartSize;
14      int m,n;
15      for (int i = 0; i < sizeLevel; i++)
```

```
16      {
17          n = ai[1] + i;
18          if ((n <= d))
19          {
20              m = ai[0] - 1;
21              if (m > 0)
22              {
23                  iNode.AddNeighbour(m_Map[m, n]);
24              }
25              m = ai[0] + sizeLevel;
26              if (m < m_Map.GetLength(0))
27              {
28                  iNode.AddNeighbour(m_Map[m, n]);
29              }
30          }
31          m = ai[0] + i;
32          if ((m <= d))
33          {
34              n = ai[1] - 1;
35              if (n > 0)
36              {
37                  iNode.AddNeighbour(m_Map[m, n]);
38              }
39              n = ai[1] + sizeLevel;
40              if (n < m_Map.GetLength(0))
41              {
42                  iNode.AddNeighbour(m_Map[m, n]);
43              }
44          }
45      }
46  }
```

Listing 8: Klasse `CQuadTree`: `SetNeighbours()`- Methode

Anhand von Größe und Position des Bereiches wird die Stelle in der Abbildungstabelle berechnet, an der sich ein Nachbar befinden kann. Falls die auf diese Art gefundene Stelle gültig ist, wird der referenzierte Bereich als Nachbar des aktuellen Bereiches gesetzt. Dabei wird ausgeschlossen, dass der gleiche Nachbar mehrfach gesetzt wird.

4.4 Implementierung der Algorithmen

Genau wie die Datenstrukturen sind auch alle Algorithmen von einer abstrakten Klasse abgeleitet. Der Aufbau der `IAlgorithm` wird im Listing 9 verdeutlicht.

```
1  public abstract class IAlgorithm
2  {
3      protected CBinaryTree<double, ISpacePart>
           m_OpenList;
4
5      protected List<ISpacePart> m_ClosedList;
6
7      protected ISpace m_Space;
8
9      public IAlgorithm(ISpace iSpace)
10     {
11         m_Space = iSpace;
12         Init();
13     }
14
15     public void Init()
16     {
17         m_OpenList = new CBinaryTree<double,
               ISpacePart>();
18         m_ClosedList = new List<ISpacePart>();
19         ISpacePart[] parts = m_Space.Parts;
20         foreach (ISpacePart p in parts)
21         {
22             if (p.DistanceTravelled >= 0)
23             {
24                 p.DistanceTravelled = double.MaxValue;
25             }
26         }
27     }
28
29     public CPath2D FindPath(CVector2D iStart,
           CVector2D iStop)
30     {
31         CPath2D p = new CPath2D();
32         p.AddPathPoint(iStop);
```

```
33          p.AddPathPath(FindPath(m_Space.
                getPartAtPosition(iStart), m_Space.
                getPartAtPosition(iStop)));
34          p.AddPathPoint(iStart);
35          return p;
36      }
37
38      protected abstract CPath2D FindPath(ISpacePart
            iStart, ISpacePart iStop);
39
40      protected CPath2D BuildPath(ISpacePart iStart,
            ISpacePart iStop)
41      {
42          CPath2D path = new CPath2D();
43          ISpacePart it = iStop.Predecessor;
44          while ((it != null) && (it != iStart))
45          {
46              path.AddPathPoint(it.Position);
47              it = it.Predecessor;
48          }
49          return path;
50      }
51
52      protected double CalcDistance(ISpacePart iStart,
            ISpacePart iStop)
53      {
54          return Math.Sqrt(Math.Pow(iStart.Position.X -
                iStop.Position.X, 2) + Math.Pow(iStart.
                Position.Y - iStop.Position.Y, 2));
55      }
56  }
```

Listing 9: Klasse `IAlgorithm`

Diese Klasse deklariert grundlegende Member und Methoden, die von jedem Algorithmus benötigt werden. Dazu zählen u.a. die `OpenList` sowie Methoden zur Erzeugung des Ergebnispfades.

4.4.1 Algorithmus von Dijkstra

Die Klasse `CDijkstra` implementiert den Algorithmus von Dijkstra. Sie erweitert die Klasse `IAlgorithm`.

```
protected override CPath2D FindPath(ISpacePart iStart,
    ISpacePart iStop)
{
    iStart.DistanceTravelled = 0;
    m_OpenList.AddNode(iStart.DistanceTravelled,
       iStart);
    ISpacePart sN;
    do
    {
        sN = m_OpenList.PopMinimalKeysValue();
        if (sN == iStop) break;
        CheckNeighbors(sN);
    } while (m_OpenList.Count > 0);

    return BuildPath(iStart, iStop);
}
```

Listing 10: Klasse `CDijkstra`: (`FindPath()`)- Methode

Die im Listing 10 gezeigte Methode bildet den Hauptbestandteil zum Finden eines Pfades durch den Raum. Dabei wird in einer Schleife stets genau der Knoten aus der `OpenList` ausgewählt, der den geringsten bereits zurückgelegten Weg aufweist. Die Schleife läuft solange, bis der Zielknoten gefunden wird oder die `OpenList` leer ist. Tritt letzterer Fall ein, dann existiert kein Pfad durch den Raum. Wenn ein Zielknoten gefunden wird, wird mithilfe der Hilfsmethode `BuildPath()` aus der Basisklasse (s. Listing 9) der Pfad aus den Daten der Vorgänger (`Predecessor`) erzeugt.

```
private void CheckNeighbours(ISpacePart iPart)
{
    ISpacePart[] neighbours = iPart.Neighbours;
    foreach (ISpacePart nb in neighbours)
    {
        if ((nb != null) && (nb.DistanceTravelled >=
            0))
        {
```

```
8                 CalcValue(iPart, nb);
9             }
10        }
11        m_ClosedList.Add(iPart);
12    }
```

Listing 11: Klasse `CDijkstra`: `CheckNeighbours()`- Methode

Diese Methode überprüft alle Nachbarn und ruft die Berechnungsmethode im Listing 12 auf. Anschließend gilt der aktuelle Knoten als untersucht und wird zur `OpenList` hinzugefügt.

```
1  private void CalcValue(ISpacePart iStart, ISpacePart
       iStop)
2  {
3      double dist = iStart.DistanceTravelled +
           CalcDistance(iStart, iStop);
4      if (dist < iStop.DistanceTravelled)
5      {
6          m_OpenList.RemoveNodeByValue(iStop);
7          iStop.DistanceTravelled = dist;
8          iStop.Predecessor = iStart;
9          m_OpenList.AddNode(iStop.DistanceTravelled,
               iStop);
10     }
11 }
```

Listing 12: Klasse `CDijkstra`: `CalcValue()`- Methode

Hier wird berechnet, wie weit der Weg vom Start- zum Nachbarschaftsknoten über den aktuellen Knoten ist. Ist der berechnete Wert kleiner als der im Nachbarschaftsknoten gespeicherte Wert, dann wird der gespeicherte Wert überschrieben. Zudem wird der Vorgänger des Nachbarschaftsknoten auf den aktuellen Knoten gesetzt. Der Nachbarschaftsknoten wird zusätzlich der `OpenList` hinzugefügt.

4.4.2 A*-Algorithmus

Die Klasse `CAStar` dient dazu, den A*-Algorithmus zu implementieren. Auch diese Klasse erbt von der abstrakten Klasse `IAlgorithm`. Ihre `FindPath()`- Methode weist den selben Aufbau auf, wie die in Listing 10 gezeigte Methode.

```
1  private void CheckNeighbors(ISpacePart iPart,
       ISpacePart iStop)
2  {
3      ISpacePart[] neighbours = iPart.Neighbours;
4      foreach (ISpacePart nb in neighbours)
5      {
6          if ((nb != null) && (nb.DistanceTravelled >=
               0))
7          {
8              CalcToGo(nb, iStop);
9              CalcValue(iPart, nb);
10         }
11     }
12     m_ClosedList.Add(iPart);
13 }
```

Listing 13: Klasse `CAStar`: `CheckNeighbours()`- Methode

Mit dieser Methode werden alle Nachbarn überprüft und die Berechnung für den zurückgelegten Weg sowie die Heuristik gestartet. Wenn der aktuelle Knoten untersucht wurde, wird er zur `ClosedList` hinzugefügt.

```
1  private void CalcValue(ISpacePart iStart, ISpacePart
       iStop)
2  {
3      double dist = iStart.DistanceTravelled +
           CalcDistance(iStart, iStop);
4      if (dist < iStop.DistanceTravelled)
5      {
6          m_OpenList.RemoveNodeByValue(iStop);
7          iStop.DistanceTravelled = dist;
8          iStop.Predecessor = iStart;
9          m_OpenList.AddNode(iStop.DistanceTravelled +
               iStop.DistanceToGo, iStop);
10     }
```

```
11 }
```

Listing 14: Klasse `CAStar`: `CalcValue()`- Methode

Hier wird zunächst die bereits zurückgelegte Distanz berechnet. Falls der dabei ermittelte Wert geringer ist als der vorhandene, wird der vorhandene Wert überschrieben. Außerdem wird der aktuelle Knoten als Vorgänger des Nachbarknotens gesetzt und zur `OpenList` hinzugefügt. Dabei bekommt er als Schlüssel die Summe aus zurückgelegtem Weg und Heuristik übergeben.

```
1 protected void CalcToGo(ISpacePart iPart, ISpacePart
    iStop)
2 {
3     iPart.DistanceToGo = CalcDistance(iPart, iStop) *
        HEURISTIK_FACTOR;
4 }
```

Listing 15: Klasse `CAStar`: `CalcToGo()`- Methode

Mit dieser Methode wird die Heuristik des Knotens berechnet. In diesem Fall wird die Luftlinie ermittelt und mit einem Faktor multipliziert. Damit ist es möglich, den Einfluss der Heuristik zu steuern, denn je höher der Faktor, desto zielgerichteter wird die Suche.

5 Untersuchung

Um die Algorithmen und Datenstrukturen zu testen, wird zunächst eine grafische Überprüfung mittels Testprogramm durchgeführt. Im Anschluss werden Zeit- und Speichermessungen vorgenommen. Für die Tests wird stets das gleiche Szenario verwendet. Die Hindernisse sowie die Position des Start- und Endpunkts werden nicht verändert. Für die grafische Überprüfung werden die Werte so ausgewählt, dass gut erkennbare Ergebnisse entstehen, damit die grundsätzlichen Unterschiede deutlich werden. Für die anderen Tests werden die Werte entsprechend vergrößert, um aussagekräftige Ergebnisse zu erzielen.

Die folgenden Abbildungen 19 bis 22 zeigen die Ausgaben des Testprogrammes, die bei Kombination der verschiedenen Datenstrukturen und Algorithmen entstanden sind. Die farbigen Quadrate stellen Hindernisse dar. Die angezeigte Linie repräsentiert den berechneten Pfad. Das blaue Gitter stellt die Datenstruktur dar, wobei die Zahlen in den Teilbereichen den Wert der Teilbereiche kennzeichnen. Der Wert einer Zelle wird vom Algorithmus bestimmt und gibt die Wegekosten vom Start an. Ein Wert von -1 bedeutet, dass der Teilbereich nicht begehbar ist. Wenn ein Teilbereich keine Zahl hat, enthält er den Ausgangswert unendlich und wurde vom Algorithmus nicht berechnet.

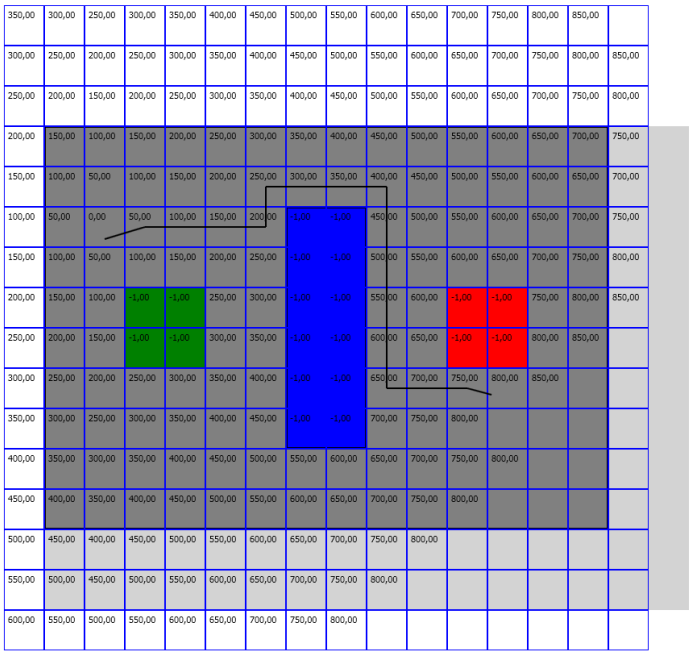

Abbildung 19: Ergebnis des Testprogramms mit der Kombination Raster und Algorithmus von Dijkstra

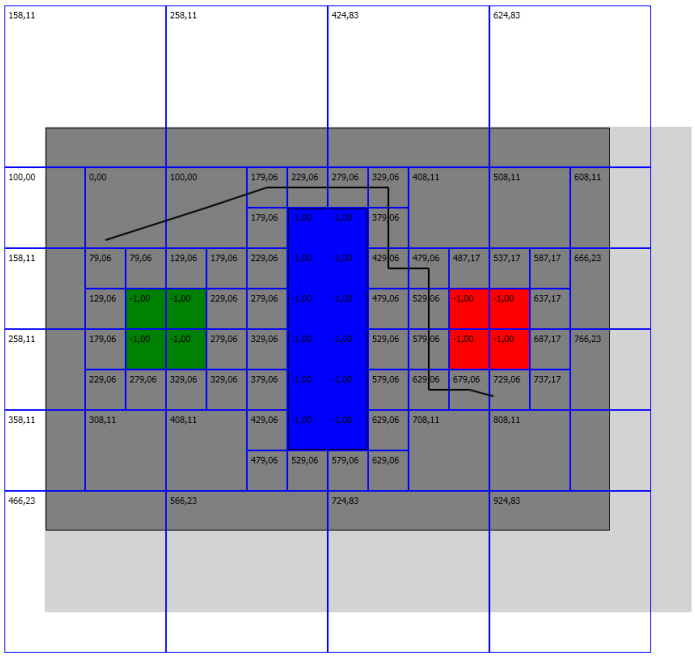

Abbildung 20: Ergebnis des Testprogramms mit der Kombination QuadTree und Algorithmus von Dijkstra

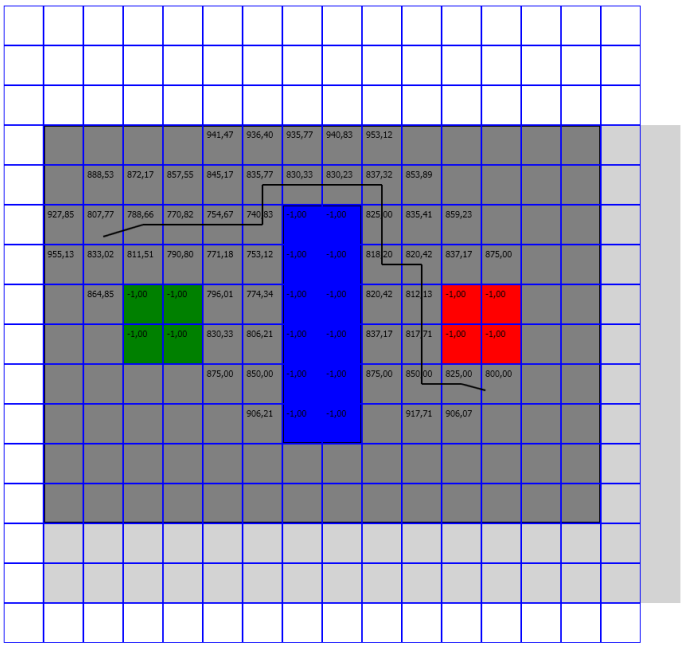

Abbildung 21: Ergebnis des Testprogramms mit der Kombination Raster und A*-Algorithmus

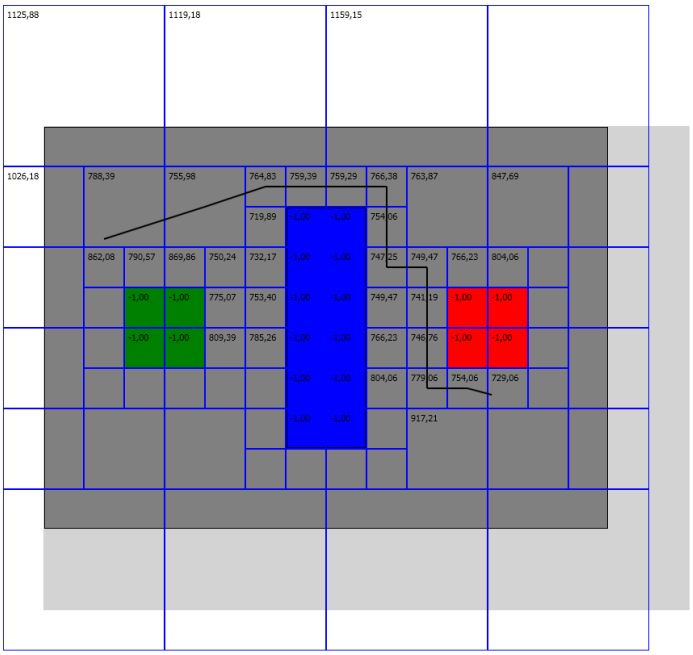

Abbildung 22: Ergebnis des Testprogramms mit der Kombination QuadTree und A*-Algorithmus

Vergleicht man Abbildung 19 mit Abbildung 21, so wird der entscheidende Vorteil des A*-Algorithmus – der Besitz einer Heuristik – deutlich: Es müssen weniger Werte berechnet werden, da der A*-Algorithmus zielgerichteter arbeitet als der Algorithmus von Dijkstra.

Bei Betrachtung der Datenstrukturen wird deutlich, dass bei Verwendung eines QuadTrees die Anzahl der Teilbereiche geringer ist. Dies hat zwei Vorteile. Zum Einen müssen weniger Daten gespeichert werden, zum Anderen muss der Algorithmus weniger Teilbereiche betrachten, wodurch sich die Rechenzeit verkürzt.

Im nächsten Schritt wird die benötigte Zeit der verschiedenen Kombinationen gemessen. Dazu baut ein separates Programm ein Testszenario auf und ermittelt die benötigte Zeit. Dabei wird schrittweise die minimale Zellgröße verringert. Dies hat zur Folge, dass die Berechnung komplexer und rechenaufwendiger wird. Die Tabelle in Abbildung 23 zeigt die Ergebnisse dieser Messung.

MinCellSize	Grid+Dijkstra	QuadTree+Dijkstra	Grid+A*	QuadTree+A*
1000	0,0625 s	0,0156 s	0,0469 s	0,0156 s
900	0,1718 s	0,0156 s	0,1406 s	0,0156 s
800	0,1718 s	0,0156 s	0,1562 s	0,0156 s
700	0,1718 s	0,0156 s	0,1406 s	0,0156 s
600	0,1718 s	0,0156 s	0,1406 s	0,0156 s
500	0,1719 s	0,0156 s	0,1562 s	0,0156 s
400	0,7812 s	0,0156 s	0,5937 s	0,0156 s
300	0,7655 s	0,0312 s	0,6093 s	0,0312 s
200	3,9685 s	0,0468 s	2,3280 s	0,0468 s
100	21,5147 s	0,1093 s	9,7183 s	0,1093 s
50	133,6667 s	0,3124 s	40,2638 s	0,3124 s

Abbildung 23: Ergebnisse der Zeittests in tabellarischer Form

Durch die schrittweise Verringerung der minimalen Zellgröße wird ein Trend erkennbar, der Aufschluss über die Leistungsfähigkeit der Kombination gibt, wie das Diagramm in Abbildung 24 verdeutlicht.

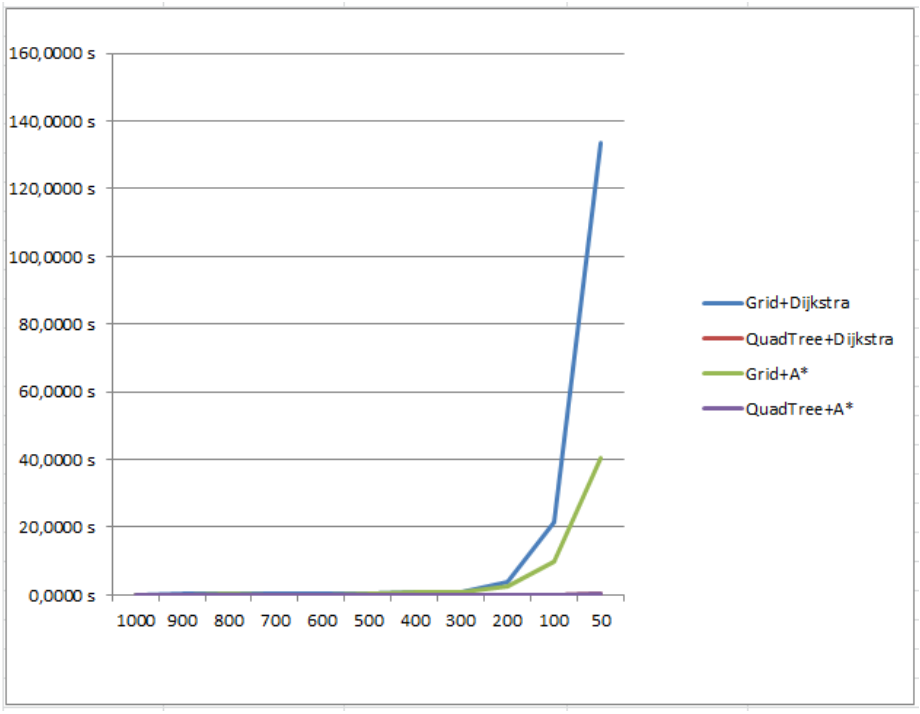

Abbildung 24: Ergebnisse der Zeittests als Diagramm

Wenn man den Verlauf der Werte zwischen den Kombinationen mit QuadTree und den Kombinationen mit Grid vergleicht, dann ist erkennbar, dass bei Verwendung eines QuadTrees eine geringere Laufzeit benötigt wird, als bei Verwen-

dung eines Grids. Diese Tatsache wird besonders bei zunehmender Komplexität deutlich. Der Unterschied zwischen den Algorithmen ist nicht so deutlich zu erkennen, da diese lediglich einen geringen Teil der Berechnung ausmachen. Daher ist bei den Werten, die in Kombination mit dem QuadTree entstanden sind, auch keine Differenz zu erkennen. Jedoch kann man bei Verwendung des Grids erkennen, dass die Laufzeit in Kombination mit dem A*-Algorithmus geringer ist, als dies in Kombination mit dem Algorithmus von Dijkstra der Fall ist.

Im Anschluss an die Tests zur benötigten Rechenzeit folgen Untersuchungen zum Speicherbedarf. Dazu wird der im Visual Studio integrierte Profiler verwendet. Mit desssen Hilfe ist es möglich, den Speicherbedarf einzelner Objekte auszulesen. Beim Test wird schrittweise die minimale Zellgröße verringert, was eine komplexere und damit größere Datenstruktur zur Folge hat. Das Augenmerk der Untersuchung liegt auf dem Speicherbedarf für alle Instanzen der jeweiligen `ISpacePart`-Klassen (`CQuadTree-` bzw. `CGridCell`). Die Tabelle in Abbildung 25 verdeutlicht die Ergebnisse dieser Messung.

MinCellSize	Grid	QuadTree
1000	40960 Byte	20016 Byte
900	163840 Byte	25544 Byte
800	163840 Byte	31920 Byte
700	163840 Byte	36912 Byte
600	163840 Byte	43440 Byte
500	163840 Byte	43440 Byte
400	655360 Byte	69168 Byte
300	655360 Byte	93360 Byte
200	2621440 Byte	145968 Byte

Abbildung 25: Ergebnisse der Speicherbedarftests als Diagramm

Auch hier wird durch die schrittweise Verringerung der minimalen Zellgröße ein Trend erkennbar, der Aufschluss darüber gibt, wie stark der Speicherbedarf bei den einzelnen Datenstrukturen steigt. Dies wird in Abbildung 26 deutlich.

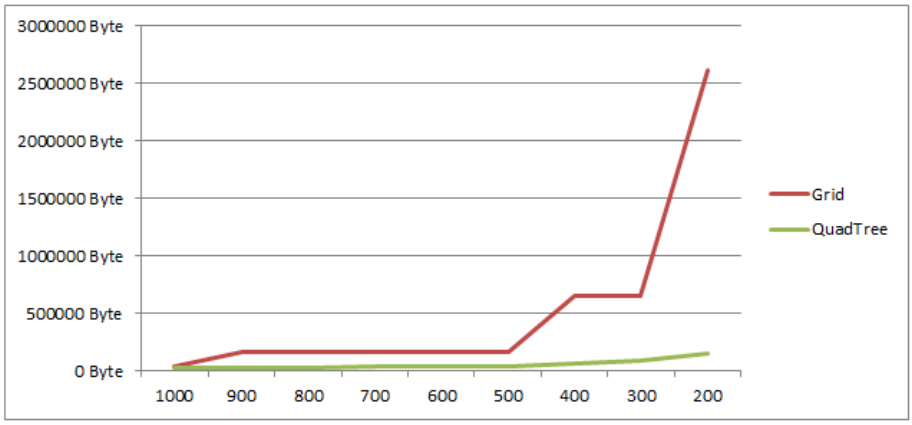

Abbildung 26: Ergebnisse der Speicherbedarftests als Diagramm

Das Diagramm zeigt, dass der Speicherbedarf des Grids schneller steigt, als beim Quad-Tree, da letzterer weniger Objekte benötigt. Im Diagramm ist zu erkennen, dass der Speicherbedarf des Grids stufenweise ansteigt. Dies ergibt sich durch die Initialisierung der Datenstruktur, wobei die Größe auf das Produkt aus minimaler Zellgröße und Zweierpotenz aufgerundet wird. Da sich für einige Werte der minimalen Zellgröße dabei die gleiche Zweierpotenz ergibt, hat das Grid in diesen Fällen gleich viele Zellen und hat damit den gleichen Speicherbedarf.

6 Ergebnis: Der Prototyp

Als Ergebnis der Untersuchung hat sich herausgestellt, dass die Kombination aus QuadTree und A*-Algorithmus am effizientesten arbeitet. Daher wird diese Kombination für die Implementierung des Prototyps verwendet. Während der Erstellung der Arbeit hat sich herausgestellt, dass sich der Arbeiter auf mehreren Ebenen bewegen kann. Daraus folgt, dass die Implementierung der Anwendung für den ema in 3D erfolgen muss. Die Datenstrukturen und Algorithmen werden aus dem Testprogramm übernommen und auf die dreidimensionale Ebene erweitert. Des Weiteren ist es notwendig, dass Bodenobjekte festgelegt werden, auf denen sich der Arbeiter bewegen kann. Es wird eine Anbindung an den Szenegraph des ema hergestellt. Den Aufbau des Prototyps zeigt das Klassendiagramm in Abbildung 27.

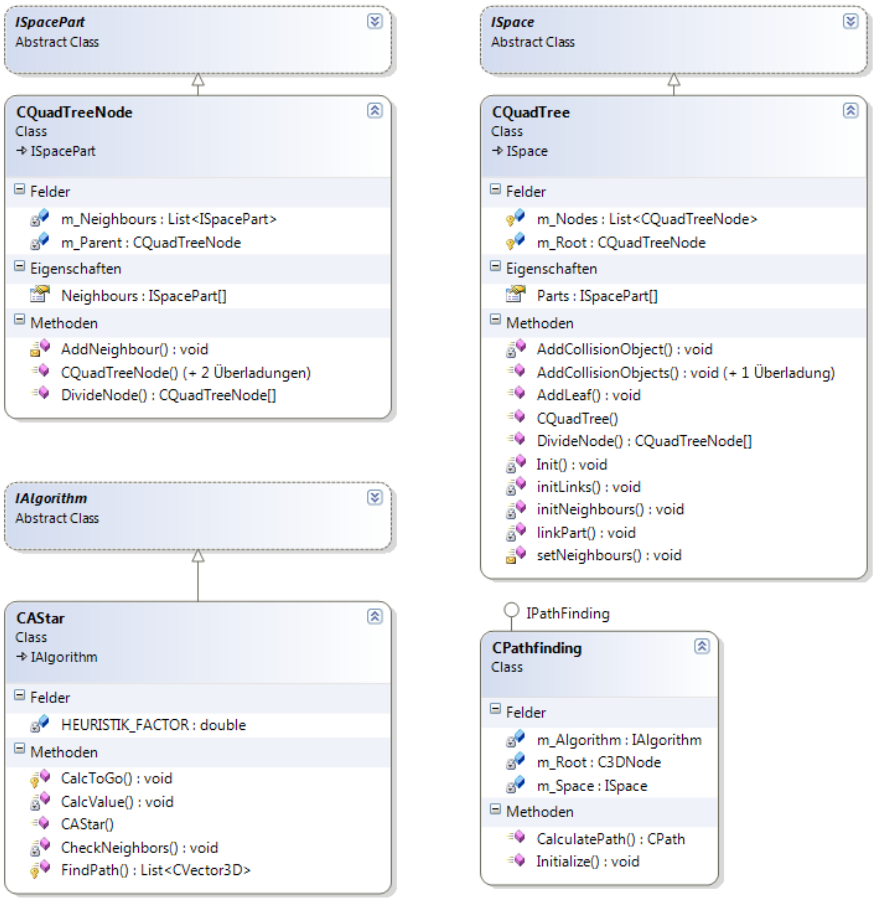

Abbildung 27: Klassendiagramm des Prototyps

Die abstrakten Klassen wurden aus dem Testsystem übernommen. Für die Klassen `CQuadTree`, `CQuadTreeNode` und `CAStar` erfolgte eine Anpassung an die ema-Umgebung. So verwenden sie den ema-Szenegraphen und nicht den Szenegraphen des Testsystems. Die Klasse `CPathFinding` implementiert das Interface `IPathFinding`, welches die Schnittstelle zum ema-Kern bildet. Sie initialisiert die Pfadfindung und führt diese aus.

Zum Testen des Systems wird der `EMACsGLViewer` verwendet, welcher von der imk automotive GmbH zum vereinfachten Darstellen des emas und der Umgebung entwickelt wurde. Abbildung 28 zeigt das Ergebnis einer Pfadsuche, das durch den `EMACsGLViewer` dargestellt wird.

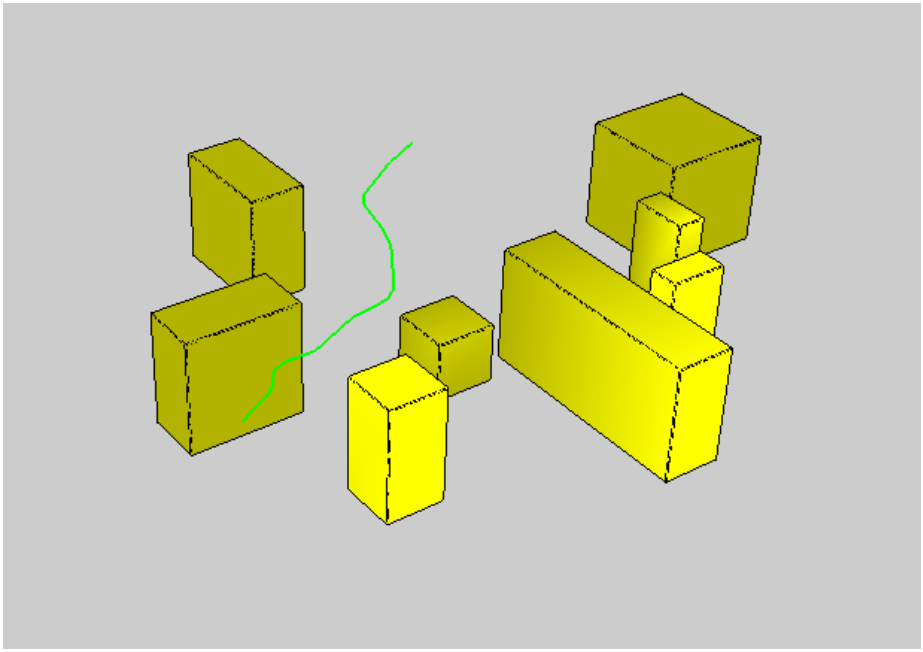

Abbildung 28: Ergebnis der Pfadsuche im `EMACsGLViewer`

Die Abbildung zeigt Objekte in Form von gelben Hüllkörpern, welche die Hindernisse repräsentieren. Die grüne Linie, welche durch die Umgebung führt, stellt den berechneten Pfad dar.

Um auch bewegliche Objekte berücksichtigen zu können, ist es notwendig, zusätzlich zu der Liste von Kollisionsobjekten auch dynamische Kollisionsobjekte einzuführen. Zudem muss die Pfadfindung um eine Zeitkomponente erweitert werden, so dass jedem Punkt auf dem Pfad eine Zeit zugewiesen werden kann. Damit ist der Algorithmus in der Lage zu ermitteln, ob ein Teilbereich zu dem

Zeitpunkt, an dem er betreten werden soll, von einem beweglichen Objekt blockiert wird oder nicht. Diese Überprüfung kann an der Stelle, an welcher der Algorithmus den Wert eines Teilbereichs berechnet, implementiert werden. Dies erfolgt analog zur Überprüfung auf Einhaltung des Mindestabstandes.

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde zunächst eine Recherche über verschiedenen Datenstrukturen zur Raumunterteilung durchgeführt sowie eine Analyse unterschiedlicher Algorithmen zur Pfadfindung vorgenommen. Die dabei als brauchbar befundenen Algorithmen und Datenstrukturen wurden für eine genauere Untersuchung in ein dafür entwickeltes Testsystem implementiert. Anhand der dabei gewonnenen Erkenntnisse wurde jeweils ein Algorithmus und eine Datenstruktur für die Entwicklung des Prototyps ausgewählt. Als Ergebnis der Arbeit ist ein Prototyp entstanden, welcher einen Raum in 3D unterteilt und dabei den kürzesten kollisionsfreien Pfad ermittelt. Der Prototyp kann über eine definierte Schnittstelle in ema integriert werden.

Als mögliche Weiterentwicklung ist es denkbar, die in Kapitel 6 dargestellten Erkenntnisse zur Berücksichtigung beweglicher Objekte programmiertechnisch umzusetzen. Des Weiteren besteht die Möglichkeit, noch weitere Algorithmen und Datenstrukturen in die Untersuchung mit einzubeziehen und ggf. zu implementieren. Ebenso ist es möglich, verschiedene Detaillierungsstufen in das Programm einzubinden, um die Performance zu verbessern.

Der Prototyp unterstützt bereits die Verwendung von Bodenobjekten. Dadurch dass die Datenstruktur in 3D umgesetzt ist, kann der ema später auch auf verschiedenen Ebenen, welche durch Rampen oder Treppen verbunden sind, arbeiten. Die Verwendung von Aufzügen und ähnlichen Verbindungsobjekten wird vom Algorithmus noch nicht verwendet, da dafür die Einführung von Portalen notwendig ist.

8 Glossar

Binary Space Partitioning-Tree Ein BSP-Tree ist eine Raumunterteilungsstruktur des n-dimensionalen Raums, bei welcher der Raum rekursiv entlang einer Teilungsgeraden bzw. Teilungsebene unterteilt wird.. 10

Digitale Fabrik Die digitale Fabrik ist eine Sammlung von Modellen und Methoden zur ganzheitlichen Planung, Realisierung, Steuerung und Verbesserung aller Prozesse der Produktion.. 4, 5

Editor menschliche Arbeit (ema) Der ema ist eine Software der imk automotive GmbH zur Planung, Simulation, Gestaltung und ergonomischen Analyse menschlicher Arbeit.. 5

European Assembly Worksheet (EAWS) Auf dem European Assembly Worksheet sind Methoden zum Bewerten physischer Belastung am Arbeitsplatz zusammengefasst.. 2

Hüllkörper (bounding volumes) Hüllkörper sind einfache geometrische Körper, welche ein komplexes geometrisches Objekt umschließen, um verschiedene Algorithmen zu beschleunigen.. 5, 6

k-dimensionaler Tree Ein kD-Tree ist eine Raumunterteilungsstruktur des k-dimensionalen Raums, bei welcher der Raum rekursiv, abwechselnd entlang einer Koordinatenachse geteilt wird.. 10

Kollisionserkennung (collision detection) Mithilfe von Kollisionserkennung ist es möglich, einander berührende oder überlappende Objekte im Raum zu erkennen.. 5, 8

Methods-Time Measurement (MTM) Unter MTM werden Methoden zum Analysieren von Arbeitsabläufen und Ermitteln von Planzeiten verstanden.. 2

Objektkoordinaten Objektkoordinaten sind relative Koordinaten eines Objektes zum übergeordneten Objekt.. 10

Oct-Tree Ein Oct-Tree ist eine Raumunterteilungsstruktur des dreidimensionalen Raums, bei welcher der Raum rekursiv in acht gleich große Teile unterteilt wird.. 10

Quad-Tree Ein Quad-Tree ist eine Raumunterteilungsstruktur des zweidimensionalen Raums, bei welcher der Raum rekursiv in vier gleich große Teile unterteilt wird.. 10

Raumunterteilung (space partioning) Mittels Raumunterteilung wird der Raum in Teilbereiche zerlegt, die sich nicht überlappen dürfen.. 5, 10

Separating Axis Theorem Das Separating Axis Theorem ist ein Algorithmus zur Erkennung von Kollisionen konvexer Körper.. 8

Szenegraph Ein Szenegraph ist eine objektorientierte Datenstruktur, welche alle Informationen einer Szene enthält.. 5, 9

Transformationsmatrix Die Transformationsmatrix ist eine Datenstruktur, die Informationen zur Translation, Rotation und Skalierung eines Objektes enthält.. 10

Verrichtungen Verrichtungen sind technologische Bewegungen und Abläufe des Werkers, in denen mehrere Grundabläufe zusammen gefasst werden.. 5

Weltkoordinaten Unter Weltkoordinaten werden absolute Koordinaten eines Objektes in der Szene verstanden.. 10

9 Verwendete Hilfsmittel

Literatur

[Ale07] Marc Alexa. *CG / CV Basis, Räumliche Datenstrukturen, Voronoi-Tesselierung / Delaunay Graph.* `PDF(s.CD)`. [Online; verfügbar am 08.09.2011, 18:52]. 2007.

[Amr09] Mario Amrehn. *Bounding Volumes and Bounding Volume Hierarchies.* `http://www10.informatik.uni-erlangen.de/Teaching/Courses/SS2009/RBD/amrehn.pdf`. [Online; verfügbar am 08.07.2011, 09:29]. 2009.

[Bau06] Sebastian Bauer. *Entwicklung einer prototypischen Software zur Planung und Visualisierung manueller Tätigkeiten.* [firmeninterne Diplomarbeit]. 2006.

[Cor+04] Thomas H. Cormen et al. *Algorithmen – eine Einführung.* Oldenbourg, München, Wien. 2004.

[Goe06] Jens-Fabian Goetzmann. *Szenegraphen.* `http://www1.informatik.uni-mainz.de/lehre/cg/SS2006_SCG/talks/SceneGraphs/scenegraphs.pdf`. [Online; verfügbar am 08.07.2011, 09:24]. 2006.

[Kas06] Daniel Kastenholz. *3D Pathfinding.* `http://wwwcg.in.tum.de/Teaching/SS2006/HauptSem/Workouts/Topic05/DATA/3dpf-doc-final.pdf`. [Online; verfügbar am 08.09.2011, 18:50]. 2006.

[SFB05] Jens Schedel, Christoph Forman, and Philipp Baumgärtel. *Kollisionserkennung.* `http://www12.cs.fau.de/edu/robertino/vortrag_robertino.pdf`. [Online; verfügbar am 08.07.2011, 09:39]. 2005.

[UFoj] Wiki der Universität Freiburg). *Wiki des Lehrstuhls für Softwaretechnik der Universität Freiburg.* `http://sopra.le-gousteau.de`. [Online; verfügbar am 08.07.2011, 09:37]. o.J.

[VR08] VDI-Richtlinien. *Digitale Fabrik Grundlagen.* [VDI-Richtlinie 4499]. 2008.

[Wikoj] DGL Wiki. *Delphi OpenGL Wiki.* `http://wiki.delphigl.com`. [Online; verfügbar am 16.05.2011, 10:15]. o.J.

Verwendete Software

- Visual Studio 2010
- Dia 0.97.1
- TexMaker 2.1

10 Anhang

Beiliegende CD

Auf der beiliegenden CD befindet sich die vorliegende Arbeit in PDF-Form. Zusätzlich enthält die CD das Visual Studio 2010 Projekt des im Laufe der Arbeit entstandenen Testprogramms und Prototyps. Ebenfalls enthalten sind einige Quellen in PDF-Format.

11 Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, den 16. September 2011

.....
Kevin Knorr